МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра программной инженерии

**ОТЧЕТ**

по лабораторным работам

по дисциплине: «Проектная деятельность»

Выполнил: Евдокимов Н.А.

Институт приборостроения, автоматизации и информационных технологий

Направление: 09.03.04 «Программная инженерия»

Группа: 71-ПГ

Проверили: Конюхова О.В., Гордиенко А.П.

Отметка о зачете:

Дата: «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2019 г.

Орёл, 2019

# содержание

[1 Структура компилятора 3](#__RefHeading___Toc2798_1434489481)

[2 Распознавание лексем 5](#__RefHeading___Toc2800_1434489481)

[3 построение списка токенов 8](#__RefHeading___Toc2802_1434489481)

[4 разбор выражений 13](#__RefHeading___Toc2804_1434489481)

[5 вычисление арифметических выражений 19](#__RefHeading___Toc2806_1434489481)

[6 разбор команд языка паскаль 20](#__RefHeading___Toc2808_1434489481)

[7 построение абстрактного синтаксического дерева 26](#__RefHeading___Toc2810_1434489481)

[8-9 интеграция лексического анализатора и синтаксического анализатора 32](#__RefHeading___Toc2812_1434489481)

# 1 Структура компилятора

**Компилятор** — это программа, которая считывает текст программы, написанной на одном языке — исходном, и транслирует его в эквивалентный текст на другом языке — целевом.

Компиляция происходит в два этапа: анализ и синтез.

Анализ разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем он использует это структуру для создания промежуточного представления исходной программы. Если анализ обнаруживает, что исходная программа неверно составлена, он должен выдать сообщение об этом. Анализ также собирает информацию об исходной программе и сохраняет ее в структуре данных, именуемой таблицей символов, которая передается вместе с промежуточным представлением синтезу.

Синтез строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов.

Компиляция представляет собой последовательность фаз, каждая из которых переводит одно из представлений программы в другое. Таблица символов, используется всеми фазами компилятора.

Некоторые компиляторы содержат фазу оптимизации. Ее назначение — преобразовать промежуточное представление так, чтобы синтез мог получить более качественную целевую программу по сравнению с той, которая может быть получена из неоптимизированного промежуточного представления. Оптимизация является необязательной фазой и может отсутствовать.

Фазы компилятора представлены на рисунке 1.

Рисунок 1 — Фазы компиляции

**Задание на лабораторную работу:** научиться использовать лексический анализатор lex.

В ходе лабораторной работы было написано следующее описание, подаваемое на вход вариации программы lex — flex.

%{

#include <stdio.h>

%}

D [0-9]

L [a-zA-Z]

ID {L}({D}|{L})\*

SPACE " "

ASSINGMENT {SPACE}\*{ID}{SPACE}\*:={SPACE}\*({ID}|{D}){SPACE}\*

%%

{ID} { printf("ID"); }

{ASSINGMENT} { printf("ASSINGMENT"); }

%%

int main(void) {

yylex();

return 0;

}

В данном описании приведены определения для целых чисел, строк, идентификаторов, пробелов и операции присвоения. Для того, чтобы передать это описание flex необходимо сохранить его в файл с расширением .l (например example.l) и выполнить следующую команду.

flex example.l

После выполнения данной команды будет сгенерирован файл lex.yy.c, который необходимо скомпилировать любым компилятором языка C. При компиляции необходимо слинковать код с библиотекой fl. Для выполнения этих задач необходимо выполнить следующую команду.

gcc lex.yy.c -lfl -o out

Результатом выполнения вышеописанной команды будет исполняемый файл out, осуществляющий лексический анализ подаваемого ему на вход текста в соответствии с заданными ранее правилами. Для его использования достаточно выполнить следующую команду (в файле src.c хранится подвергаемый анализу текст).

cat src.c | ./out

# 2 Распознавание лексем

**Лексема** — это последовательность машинных символов исходного кода программы, имеющие определенное совокупное значение.

Распознание лексем осуществляется путем использования лексического анализа, выполняемого лексическим анализатором. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности — лексемы.

Для каждой лексемы анализатор строит выходной токен, представляющий собой пару: имя токена, значение атрибута. Имя токена — это абстрактный символ, использующийся во время синтаксического анализа, а значение атрибута указывает на запись в таблице символов, соответствующую этому токену.

**Задание на лабораторную работу:** определить введенную лексику на принадлежность классам: целое число, число с плавающей запятой, идентификатор.

Для выполнения лабораторной работы была написана программа, читающая последовательность токенов и определяющая тип каждого токена путем использования регулярных выражений. Код программы представлен ниже.

#include <iostream>

#include <fstream>

#include <regex>

#include <map>

using namespace std;

enum TYPES {

ID, CONST\_INT, CONST\_FLOAT, ERROR

};

map<TYPES, string> typeToString = {

{ID, "ID"},

{CONST\_INT, "CONST\_INT"},

{CONST\_FLOAT, "CONST\_FLOAT"},

{ERROR, "ERROR"}

};

ifstream inStream;

string yytext;

TYPES currentType;

bool isId(const string& token) {

bool isId = true;

regex pattern(R"([a-zA-Z](\d|[a-zA-Z])\*)");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isId = false;

}

return isId;

}

bool isConstInt(const string& token) {

bool isConstInt = true;

regex pattern(R"(-?\d+)");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isConstInt = false;

}

return isConstInt;

}

bool isConstFloat(const string& token) {

bool isConstFloat = true;

regex pattern(R"((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isConstFloat = false;

}

return isConstFloat;

}

void yylex() {

if (inStream.eof()) {

cout << "Reached end of file!" << endl;

return;

}

string token;

inStream >> token;

string s = " s = asd";

if (token.length() == 1 && token[0] == '\n') {

return;

}

if (isId(token)) {

yytext = token;

currentType = ID;

} else if (isConstInt(token)) {

yytext = token;

currentType = CONST\_INT;

} else if (isConstFloat(token)) {

yytext = token;

currentType = CONST\_FLOAT;

} else {

yytext = token;

currentType = ERROR;

}

}

int main(int argc, char\* argv[]) {

if (argc <= 1) {

cout << "Usage: ./lab2 <file\_to\_parse>";

exit(0);

}

try {

inStream.open(argv[1]);

} catch(const ifstream::failure& e) {

cout << "Can't open file: " << argv[1];

exit(0);

}

yylex();

cout << typeToString[currentType] << ':' << yytext;

switch (currentType) {

case CONST\_INT: {

int i = stoi(yytext);

break;

}

case CONST\_FLOAT: {

float i = stof(yytext);

break;

}

default: {

string i = yytext;

}

}

return 0;

}

# 3 построение списка токенов

**Список** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза.

Список, как правило, определяется в связной памяти и предоставляет последовательный доступ к своим элементам, что исключает возможность прямого доступа, как при использовании массивов. В случае односвязного списка элементом списка является запись, содержащая сами данные и указатель на следующий элемент.

**Задание на лабораторную работу:** на основе предыдущей лабораторной работы написать программу, формирующую список токенов.

Для выполнения лабораторной работы была реализована такая структура данных, как односвязный список. Его исходный код содержится в файле LinkedList.h и приведен ниже. Реализация содержит методы для добавления, доступа и удаления элементов.

template <class T>

class LinkedList {

struct ITEM {

T\* data;

ITEM\* next = nullptr;

} item;

ITEM\* root = nullptr;

long length = 0;

public:

void addItem(T\* newItem) {

if (root == nullptr) {

root = new ITEM();

root->data = newItem;

length++;

return;

}

ITEM\* currentItem = root;

while (currentItem->next != nullptr) {

currentItem = currentItem->next;

}

currentItem->next = new ITEM();

currentItem = currentItem->next;

currentItem->data = newItem;

length++;

}

T\* getItem(int index) {

if (index + 1 > length || index < 0) {

return nullptr;

}

ITEM\* currentItem = root;

for (int i = 0; i < index; i++) {

currentItem = currentItem->next;

}

return currentItem->data;

}

void deleteItem(int index) {

if (index + 1 > length || index < 0) {

return;

}

ITEM\* currentItem = root;

for (int i = 0; i < index - 1; i++) {

currentItem = currentItem->next;

}

ITEM\* temp = currentItem->next;

currentItem->next = currentItem->next->next;

delete temp->data;

delete temp;

length--;

}

};

Помимо односвязного списка были реализованы следующие структуры (реализация каждой из них приведена в файле structs.h).

enum TYPES {

ID, CONST\_INT, CONST\_FLOAT, ERROR

};

enum SYMBOL\_CLASS {

NUMBER, OPERATION, BRACKET

};

union SYMBOL\_VALUE {

char special\_char;

float num;

};

struct TOKEN {

SYMBOL\_CLASS type;

SYMBOL\_VALUE sym;

};

Далее приведен исходный код программы, реализующей построение списка токенов.

#include <iostream>

#include <fstream>

#include <regex>

#include <map>

#include "structs.h"

#include "LinkedList.h"

using namespace std;

map<SYMBOL\_CLASS, string> typeToString = {

{NUMBER, "NUMBER"},

{OPERATION, "OPERATION"},

{BRACKET, "BRACKET"}

};

ifstream inStream;

LinkedList<TOKEN> list;

bool isConstInt(const string& token) {

bool isConstInt = true;

regex pattern(R"(-?\d+)");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isConstInt = false;

}

return isConstInt;

}

bool isOperation(const string &token) {

bool isOperation = true;

regex pattern(R"([+-/\*])");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isOperation = false;

}

return isOperation;

}

bool isBracket(const string &token) {

return token.length() == 1 && (token[0] == '(' || token[0] == ')');

}

bool isConstFloat(const string& token) {

bool isConstFloat = true;

regex pattern(R"((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isConstFloat = false;

}

return isConstFloat;

}

void yylex() {

if (inStream.eof()) {

cout << "Reached end of file!" << endl;

return;

}

string tokenStr;

inStream >> tokenStr;

if (tokenStr.length() == 1 && tokenStr[0] == '\n') {

return;

}

TOKEN\* token = new TOKEN();

if (isConstFloat(tokenStr) || isConstInt(tokenStr)) {

token->type = SYMBOL\_CLASS::NUMBER;

token->sym.num = stof(tokenStr);

} else if (isOperation(tokenStr)) {

token->type = SYMBOL\_CLASS::OPERATION;

token->sym.special\_char = tokenStr[0];

} else if (isBracket(tokenStr)) {

token->type = SYMBOL\_CLASS::BRACKET;

token->sym.special\_char = tokenStr[0];

}

list.addItem(token);

}

int main(int argc, char\* argv[]) {

if (argc <= 1) {

cout << "Usage: ./lab2 <file\_to\_parse>";

exit(0);

}

try {

inStream.open(argv[1]);

} catch(const ifstream::failure& e) {

cout << "Can't open file: " << argv[1];

exit(0);

}

{

int i = 0;

while (!inStream.eof()) {

yylex();

TOKEN\* token = list.getItem(i);

cout << typeToString[token->type] << ':';

if (token->type == NUMBER) {

cout << token->sym.num << endl;

} else {

cout << token->sym.special\_char << endl;

}

i++;

}

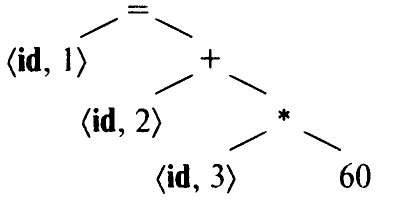
}

return 0;

}

# 4 разбор выражений

Разбор выражений выполняется в фазе синтаксического анализа. Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Как правило, для выполнения такой задачи используется синтаксическое дерево, в котором каждый узел представляет собой операцию, а дочерние узлы — аргументы этой операции. Пример синтаксического дерева изображен на рисунке 2.

Рисунок 2 — Пример синтаксического дерева

**Задание на лабораторную работу:** сформировать дерево выражения и вывести его.

Для выполнения данной лабораторной работы был реализован класс AstBuilder, осуществляющий формирование дерева выражения. Данная операция выполняется в два этапа: перевод выражения в обратную польскую нотацию и построение дерева на основе результата предыдущего преобразования.

Исходный код AstBuilder представлен ниже.

#include "AstBuilder.h"

#include <stack>

#include <stdexcept>

AstBuilder::AstBuilder() {

classToPriority[OPERATION\_ADD] = 1;

classToPriority[OPERATION\_SUB] = 1;

classToPriority[OPERATION\_MUL] = 2;

classToPriority[OPERATION\_DIV] = 2;

classToPriority[OPEN\_BRACKET] = 0;

}

vector<TOKEN> AstBuilder::buildRpn(vector<TOKEN> &expression) {

// TOKEN currentToken;

vector<TOKEN> result;

stack<TOKEN> st;

for (TOKEN currentToken : expression) {

switch (currentToken.type) {

case SYMBOL\_CLASS::NUMBER: {

result.push\_back(currentToken);

break;

}

case OPEN\_BRACKET: {

st.push(currentToken);

break;

}

case CLOSE\_BRACKET: {

while (st.top().type != OPEN\_BRACKET) {

result.push\_back(st.top());

st.pop();

if (st.empty()) {

throw out\_of\_range("Invalid expression");

}

}

st.pop();

break;

}

default: {

while (!st.empty() && (classToPriority[currentToken.type] <= classToPriority[st.top().type])) {

result.push\_back(st.top());

st.pop();

}

st.push(currentToken);

}

}

}

while (!st.empty()) {

result.push\_back(st.top());

st.pop();

}

return result;

}

AstNode\* AstBuilder::getLastLeftFree(AstNode \*root) {

AstNode\* result = root;

AstNode\* currentNode = root;

while (true) {

if (currentNode->left == nullptr) {

result = currentNode;

}

if (currentNode->right != nullptr && currentNode->right->token.type != SYMBOL\_CLASS::NUMBER) {

currentNode = currentNode->right;

}

else {

break;

}

}

return result;

}

AstNode\* AstBuilder::buildAstTree(vector<TOKEN> expression) {

vector<TOKEN> rpn = buildRpn(expression);

stack<AstNode\*> stOperations;

AstNode\* root = new AstNode();

root->token = rpn.back();

AstNode\* prevNode = root;

for (long long i = rpn.size() - 2; i >= 0; i--) {

AstNode\* currentNode = new AstNode();

currentNode->token = rpn[i];

while (prevNode->left != nullptr && prevNode->right != nullptr) {

prevNode = stOperations.top();

stOperations.pop();

}

if (prevNode->right == nullptr) {

prevNode->right = currentNode;

}

else {

prevNode->left = currentNode;

}

if (rpn[i].type != SYMBOL\_CLASS::NUMBER) {

stOperations.push(prevNode);

prevNode = currentNode;

}

}

return root;

}

Для вывода дерева была реализована процедура print\_tree, принимающая на вход корень дерева и уровень. Второй аргумент используется для вывода нужного количества отступов. Данная процедура имеет рекурсивную природу. Ее исходный код приведен далее.

void print\_Tree(AstNode\* p, int level) {

if (p) {

if (p->right) print\_Tree(p->right, level + 4);

if (level) {

cout << setw(level) << ' ';

}

if (p->token.type == SYMBOL\_CLASS::NUMBER) {

cout << p->token.sym.num << "\n";

} else {

cout << p->token.sym.special\_char << "\n";

}

if (p->left) print\_Tree(p->left, level + 4);

}

}

Ниже приведен исходный код программы, осуществляющей построение дерева выражения.

1. #include <iostream>

#include <fstream>

#include <regex>

#include <map>

#include<iomanip>

#include "structs.h"

#include "LinkedList.h"

#include "AstBuilder.h"

using namespace std;

map<SYMBOL\_CLASS, string> typeToString = {

{NUMBER, "NUMBER"},

{OPERATION\_ADD, "OPERATION\_ADD"},

{OPERATION\_DIV, "OPERATION\_DIV"},

{OPERATION\_MUL, "OPERATION\_MUL"},

{OPERATION\_SUB, "OPERATION\_SUB"},

{CLOSE\_BRACKET, "CLOSE\_BRACKET"},

{OPEN\_BRACKET, "OPEN\_BRACKET"}

};

ifstream inStream;

LinkedList<TOKEN> list;

bool isConstInt(const string& token) {

bool isConstInt = true;

regex pattern(R"(-?\d+)");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isConstInt = false;

}

return isConstInt;

}

bool isOperation(const string &token) {

bool isOperation = true;

regex pattern(R"([+-/\*])");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isOperation = false;

}

return isOperation;

}

bool isBracket(const string &token) {

return token.length() == 1 && (token[0] == '(' || token[0] == ')');

}

bool isConstFloat(const string& token) {

bool isConstFloat = true;

regex pattern(R"((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))");

smatch match;

regex\_match(token, match, pattern);

if (match.length() <= 0) {

isConstFloat = false;

}

return isConstFloat;

}

void yylex() {

if (inStream.eof()) {

cout << "Reached end of file!" << endl;

return;

}

string tokenStr;

inStream >> tokenStr;

if (tokenStr.length() == 1 && tokenStr[0] == '\n') {

return;

}

TOKEN\* token = new TOKEN();

if (isConstFloat(tokenStr) || isConstInt(tokenStr)) {

token->type = SYMBOL\_CLASS::NUMBER;

token->sym.num = stof(tokenStr);

} else if (isOperation(tokenStr)) {

switch (tokenStr[0]) {

case '\*':

token->type = SYMBOL\_CLASS::OPERATION\_MUL;

break;

case '/':

token->type = SYMBOL\_CLASS::OPERATION\_DIV;

break;

case '+':

token->type = SYMBOL\_CLASS::OPERATION\_ADD;

break;

case '-':

token->type = SYMBOL\_CLASS::OPERATION\_SUB;

break;

}

token->sym.special\_char = tokenStr[0];

} else if (isBracket(tokenStr)) {

switch (tokenStr[0]) {

case '(':

token->type = SYMBOL\_CLASS::OPEN\_BRACKET;

break;

case ')':

token->type = SYMBOL\_CLASS::CLOSE\_BRACKET;

break;

}

token->sym.special\_char = tokenStr[0];

}

list.addItem(token);

}

void print\_Tree(AstNode\* p, int level) {

if (p) {

if (p->right) print\_Tree(p->right, level + 4);

if (level) {

cout << setw(level) << ' ';

}

if (p->token.type == SYMBOL\_CLASS::NUMBER) {

cout << p->token.sym.num << "\n";

} else {

cout << p->token.sym.special\_char << "\n";

}

if (p->left) print\_Tree(p->left, level + 4);

}

}

int main(int argc, char\* argv[]) {

if (argc <= 1) {

cout << "Usage: ./lab2 <file\_to\_parse>";

exit(0);

}

try {

inStream.open(argv[1]);

} catch(const ifstream::failure& e) {

cout << "Can't open file: " << argv[1];

exit(0);

}

{

int i = 0;

while (!inStream.eof()) {

yylex();

TOKEN\* token = list.getItem(i);

i++;

}

vector<TOKEN> tokenVector;

for (unsigned long j = 0; j < list.size(); j++) {

tokenVector.push\_back(\*list.getItem(j));

}

AstBuilder builder = AstBuilder();

AstNode\* astRoot = builder.buildAstTree(tokenVector);

print\_Tree(astRoot, 0);

cout << endl << endl;

cout << "Calculation result: " << calculateTree(astRoot);

}

return 0;

}

# 5 вычисление арифметических выражений

Арифметическое выражение содержит только арифметические операторы и операнды. Как правило, выражения подлежащие вычислению представлены в виде древовидной структуры данных, у которой каждый узлы представляют операторы, а дочерние узлы — операнды.

**Задание на лабораторную работу:**  на основе предыдущей лабораторной работы написать программу, осуществляющую вычисление арифметического выражения на основе дерева выражения.

Для выполнения лабораторной работы была написана рекурсивная функция calculateTree, принимающая в качестве аргумента узел дерева. В качестве возвращаемого значения — результат вычислений. Весь остальной исходный код соответствует предыдущей лабораторной работе.

float calculateTree(AstNode\* root) {

float result = 0;

switch (root->token.type)

{

case OPERATION\_ADD:

result = calculateTree(root->left) + calculateTree(root->right);

break;

case OPERATION\_SUB:

result = calculateTree(root->left) - calculateTree(root->right);

break;

case OPERATION\_MUL:

result = calculateTree(root->left) \* calculateTree(root->right);

break;

case OPERATION\_DIV:

result = calculateTree(root->left) / calculateTree(root->right);

break;

case NUMBER:

result = root->token.sym.num;

break;

default:

break;

}

return result;

}

# 6 разбор команд языка паскаль

При трансляции появляется необходимость сгруппировать отдельные токены в команды языка. При этом важно учесть семантику языка. Если команда построена неправильно, дальнейшие преобразования невозможны так как написанный код содержит ошибки. При трансляции данный процесс является основой для построения синтаксического дерева.

**Задание на лабораторную работу:** написать программу, осуществляющую разбор команд языка Паскаль.

Для выполнения лабораторной работы был введен перечислимый тип, содержащий названия рассматриваемых в рамках работы кострукций. Исходный код данного перечисления представлен ниже.

enum GRAMMAR\_SYMBOL\_TYPE {

WHILE, IF, ELSE, ASSIGNMENT, ERROR, BLOCK

};

Далее представлен исходный код программы, проверяющей правильность написанного на языке Паскаль кода. Основная работа происходит в рекурсивной функции checkSyntax, принимающей список строк и возвращающей логический тип — правильно ли составлен код. Если функция находит сложную конструкцию, состоящую более, чем из одной строчки, она составляет подсписок, состоящий из операций данной конструкции, после чего происходит вызов этой же функции с составленным подсписком.

#include <iostream>

#include <fstream>

#include <vector>

#include <regex>

#include "grammar\_symbols.h"

#include "main.h"

#define PASCAL\_BEGIN "BEGIN"

#define PASCAL\_END "END"

#define PASCAL\_ELSE "ELSE"

#define PASCAL\_IF "IF"

#define PASCAL\_WHILE "WHILE"

#define PASCAL\_ASSIGNMENT ":="

using namespace std;

bool doesExpressionReturnsSomething(const string& line) {

regex pattern(R"(^(\s\*(BE|ID)|(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*)$)");

smatch match;

if (regex\_match(line, match, pattern)) {

return true;

}

int i = 0/2;

string patternStr = R"(^\s\*(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*[+-/\*%]\s\*(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))$)";

pattern = regex(patternStr, regex\_constants::icase);

return regex\_match(line, match, pattern);

}

bool isAssignmentCorrect(const string& line) {

string patternStr = R"(\s\*ID\s\*:=\s\*BE\s\*)";

regex patternSimpleAssignment(patternStr);

smatch match;

bool isAssignmentCorrect = regex\_match(line, match, patternSimpleAssignment);

if (!isAssignmentCorrect) {

patternStr = R"(\s\*ID\s\*:=\s\*(ID|(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*)\s\*([+-/\*]?\s\*(ID|(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*)\s\*)\*\s\*$)";

regex pattern(patternStr);

isAssignmentCorrect = regex\_match(line, match, pattern);

}

cout << "Is assignment correct: " << isAssignmentCorrect <<" string: '" << line << '\'' << endl;

return isAssignmentCorrect;

}

bool isIfCorrect(const string& line) {

string patternStr = R"(^\s\*IF\s\*BE\s\*THEN\s\*$)";

regex pattern(patternStr);

smatch match;

bool isIfCorrect = regex\_match(line, match, pattern);

cout << "Is IF correct: " << isIfCorrect <<" string: '" << line << '\'' << endl;

return isIfCorrect;

}

bool isElseCorrect(const string& line) {

string patternStr = R"(^\s\*ELSE\s\*$)";

regex pattern(patternStr);

smatch match;

bool isElseCorrect = regex\_match(line, match, pattern);

cout << "Is ELSE correct: " << isElseCorrect << " string: '" << line << '\'' << endl;

return isElseCorrect;

}

bool isWhileCorrect(const string& line) {

string patternStr = R"(^\s\*WHILE\s\*BE\s\*DO\s\*$)";

regex pattern(patternStr);

smatch match;

bool isWhileCorrect = regex\_match(line, match, pattern);

cout << "Is WHILE correct: " << isWhileCorrect <<" string: '" << line << '\'' << endl;

return isWhileCorrect;

}

GRAMMAR\_SYMBOL\_TYPE determineGrammarSymbolType(string& line) {

// block

if (line.find(PASCAL\_BEGIN) != string::npos) {

return GRAMMAR\_SYMBOL\_TYPE::BLOCK;

}

// Assignment

if (line.find(PASCAL\_ASSIGNMENT) != string::npos) {

bool isCorrect = isAssignmentCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::ASSIGNMENT : GRAMMAR\_SYMBOL\_TYPE::ERROR;

}

// if

if (line.find(PASCAL\_IF) != string::npos) {

bool isCorrect = isIfCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::IF : GRAMMAR\_SYMBOL\_TYPE::ERROR;

}

// else

if (line.find(PASCAL\_ELSE) != string::npos) {

bool isCorrect = isElseCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::ELSE : GRAMMAR\_SYMBOL\_TYPE::ERROR;

}

// while

if (line.find(PASCAL\_WHILE) != string::npos) {

bool isCorrect = isWhileCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::WHILE : GRAMMAR\_SYMBOL\_TYPE::ERROR;

}

return GRAMMAR\_SYMBOL\_TYPE::ERROR;

}

bool isOneLiner(const string& line) {

return line.find(PASCAL\_ASSIGNMENT) != string::npos;

}

void makeExpression(const vector<string>& parentExpression, size\_t index, vector<string>& returnVector) {

if (parentExpression[index].find(PASCAL\_BEGIN) != string::npos) {

returnVector.push\_back(parentExpression[index]);

int foundBegins = 1;

index++;

while (index != parentExpression.size() && foundBegins != 0) {

if (parentExpression[index].find(PASCAL\_BEGIN) != string::npos) {

foundBegins++;

}

if (parentExpression[index].find(PASCAL\_END) != string::npos) {

foundBegins--;

}

returnVector.push\_back(parentExpression[index]);

index++;

}

return;

} else if (isOneLiner(parentExpression[index])) {

returnVector.push\_back(parentExpression[index]);

return;

} else {

returnVector.push\_back(parentExpression[index]);

return makeExpression(parentExpression, index + 1, returnVector);

}

}

bool checkSyntax(const vector<string>& lines) {

bool isCorrect = true;

size\_t i = 0;

while (i < lines.size()) {

string currentLine = lines[i];

GRAMMAR\_SYMBOL\_TYPE currentType = determineGrammarSymbolType(currentLine);

switch (currentType) {

case GRAMMAR\_SYMBOL\_TYPE::ERROR: {

return false;

}

case GRAMMAR\_SYMBOL\_TYPE::BLOCK: {

// there is BEGIN but no END

if (lines[lines.size() - 1].find(PASCAL\_END) == string::npos) {

return false;

}

vector<string> expression;

for (size\_t j = 1; j <= lines.size() - 2; j++) {

expression.push\_back(lines[j]);

}

isCorrect = checkSyntax(expression);

if (!isCorrect) {

return false;

}

i += expression.size() + 1;

break;

}

case GRAMMAR\_SYMBOL\_TYPE::IF: {

vector<string> expression;

makeExpression(lines, i + 1, expression);

isCorrect = checkSyntax(expression);

if (!isCorrect) { return false; }

i += expression.size();

if (lines.size() > (i + 1)) {

string nextString = lines[i + 1];

if (determineGrammarSymbolType(nextString) == GRAMMAR\_SYMBOL\_TYPE::ELSE) {

i++;

expression.clear();

makeExpression(lines, i + 1, expression);

isCorrect = checkSyntax(expression);

if (!isCorrect) { return false; }

i += expression.size();

}

}

break;

}

case GRAMMAR\_SYMBOL\_TYPE::ELSE: {

return false;

break;

}

case GRAMMAR\_SYMBOL\_TYPE::WHILE: {

vector<string> expression;

makeExpression(lines, i + 1, expression);

isCorrect = checkSyntax(expression);

if (!isCorrect) { return false; }

i += expression.size();

break;

}

default: {}

}

i++;

}

return isCorrect;

}

int main() {

string filePath = "test";

ifstream input(filePath);

if (!input.good()) {

cout << "Can't open source file" << endl;

exit(-1);

}

vector<string> lines;

string currentLine;

while (getline(input, currentLine)) {

if (!currentLine.empty()) {

lines.push\_back(currentLine);

}

}

bool isCorrect = checkSyntax(lines);

if (isCorrect) {

cout << "Source code is correct" << endl;

} else {

cout << "Source code is not correct" << endl;

}

return 0;

}

# 7 построение абстрактного синтаксического дерева

Одним из промежуточных представлений исходного кода при его трансляции является синтаксическое дерево. Оно содержит информацию об операторах языка и их операндах. Так, у оператора присвоения два операнда: идентификатор, которому присваивается новое значение, и само значение; у оператора ветвления три операнда: логическое выражение и две ветки; у оператора while два операнда: логическое выражение и команды. Также следует выделить такой тип как блок кода. Он содержит в себе список команд, каждый элемент которого может быть корнем нового дерева.

**Задание на лабораторную работу:** на основе предыдущей лабораторной работы написать программу, осуществляющую построение абстрактного синтаксического дерева.

Для выполнения лабораторной работы был переиспользован класс AstBuilder с доработками касательно построения дерева. Его исходный код представлен ниже.

#include "AstBuilder.h"

std::string& ltrim(std::string& str, const std::string& chars = "\t\n\v\f\r ")

{

str.erase(0, str.find\_first\_not\_of(chars));

return str;

}

std::string& rtrim(std::string& str, const std::string& chars = "\t\n\v\f\r ")

{

str.erase(str.find\_last\_not\_of(chars) + 1);

return str;

}

std::string& trim(std::string& str, const std::string& chars = "\t\n\v\f\r ")

{

return ltrim(rtrim(str, chars), chars);

}

bool AstBuilder::checkAndBuildTree(vector<string> lines, AstNode\* &root)

{

bool isCorrect = true;

size\_t i = 0;

while (i < lines.size()) {

AstNode\* result = new AstNode();

if (root != nullptr) {

root->list.push\_back(result);

}

else {

root = result;

}

string currentLine = lines[i];

GRAMMAR\_SYMBOL\_TYPE currentType = determineGrammarSymbolType(currentLine);

switch (currentType) {

case GRAMMAR\_SYMBOL\_TYPE::*ERROR*: {

return false;

}

case GRAMMAR\_SYMBOL\_TYPE::*BLOCK*: {

// there is BEGIN but no END

if (lines[lines.size() - 1].find(PASCAL\_END) == string::npos) {

return false;

}

vector<string> expression;

for (size\_t j = 1; j <= lines.size() - 2; j++) {

expression.push\_back(lines[j]);

}

root->type = TYPE\_NODE::*NODE\_BLOCK*;

isCorrect = checkAndBuildTree(expression, root);

if (!isCorrect) {

return false;

}

i += expression.size() + 1;

break;

}

case GRAMMAR\_SYMBOL\_TYPE::*IF*: {

vector<string> expression;

makeExpression(lines, i + 1, expression);

result->type = TYPE\_NODE::*NODE\_IF*;

AstNode\* be = new AstNode();

be->data = "BE";

result->list.push\_back(be);

result->list.push\_back(nullptr);

isCorrect = checkAndBuildTree(expression, result->list.back());

if (!isCorrect) { return false; }

i += expression.size();

if (lines.size() > (i + 1)) {

string nextString = lines[i + 1];

if (determineGrammarSymbolType(nextString) == GRAMMAR\_SYMBOL\_TYPE::*ELSE*) {

i++;

expression.clear();

makeExpression(lines, i + 1, expression);

result->list.push\_back(nullptr);

isCorrect = checkAndBuildTree(expression, result->list.back());

if (!isCorrect) { return false; }

i += expression.size();

}

}

break;

}

case GRAMMAR\_SYMBOL\_TYPE::*ELSE*: {

return false;

break;

}

case GRAMMAR\_SYMBOL\_TYPE::*WHILE*: {

vector<string> expression;

makeExpression(lines, i + 1, expression);

result->type = TYPE\_NODE::*NODE\_WHILE*;

AstNode\* be = new AstNode();

be->data = "BE";

result->list.push\_back(be);

result->list.push\_back(nullptr);

isCorrect = checkAndBuildTree(expression, result->list.back());

if (!isCorrect) { return false; }

i += expression.size();

break;

}

case GRAMMAR\_SYMBOL\_TYPE::*ASSIGNMENT*: {

result->type = TYPE\_NODE::*NODE\_ASSIGNMENT*;

AstNode\* data = new AstNode();

data->data = "ID";

result->list.push\_back(data);

data = new AstNode();

size\_t pos = currentLine.find(PASCAL\_ASSIGNMENT);

string rightPart = currentLine.substr(pos + 2);

trim(rightPart);

data->data = rightPart;

result->list.push\_back(data);

break;

}

default: {}

}

i++;

}

return isCorrect;

}

void AstBuilder::deleteTree(AstNode\* root)

{

if (root == nullptr) {

return;

}

for (AstNode\* node : root->list) {

deleteTree(node);

}

delete root;

}

AstBuilder::AstBuilder()

{

}

AstNode\* AstBuilder::buildAstTree(vector<string> lines)

{

AstNode\* result = nullptr;

if (!checkAndBuildTree(lines, result)) {

deleteTree(result);

result = nullptr;

}

return result;

}

bool AstBuilder::doesExpressionReturnsSomething(const string& line)

{

regex pattern(R"(^(\s\*(BE|ID)|(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*)$)");

smatch match;

if (regex\_match(line, match, pattern)) {

return true;

}

int i = 0/2;

string patternStr = R"(^\s\*(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*[+-/\*%]\s\*(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))$)";

pattern = regex(patternStr, regex\_constants::icase);

return regex\_match(line, match, pattern);

}

bool AstBuilder::isAssignmentCorrect(const string& line)

{

string patternStr = R"(\s\*ID\s\*:=\s\*BE\s\*)";

regex patternSimpleAssignment(patternStr);

smatch match;

bool isAssignmentCorrect = regex\_match(line, match, patternSimpleAssignment);

if (!isAssignmentCorrect) {

patternStr = R"(\s\*ID\s\*:=\s\*(ID|(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*)\s\*([+-/\*]?\s\*(ID|(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+))|ID|((-?\d+)))\s\*)\s\*)\*\s\*$)";

regex pattern(patternStr);

isAssignmentCorrect = regex\_match(line, match, pattern);

}

cout << "Is assignment correct: " << isAssignmentCorrect <<" string: '" << line << '\'' << endl;

return isAssignmentCorrect;

}

bool AstBuilder::isIfCorrect(const string& line)

{

string patternStr = R"(^\s\*IF\s\*BE\s\*THEN\s\*$)";

regex pattern(patternStr);

smatch match;

bool isIfCorrect = regex\_match(line, match, pattern);

cout << "Is IF correct: " << isIfCorrect <<" string: '" << line << '\'' << endl;

return isIfCorrect;

}

bool AstBuilder::isElseCorrect(const string& line)

{

string patternStr = R"(^\s\*ELSE\s\*$)";

regex pattern(patternStr);

smatch match;

bool isElseCorrect = regex\_match(line, match, pattern);

cout << "Is ELSE correct: " << isElseCorrect << " string: '" << line << '\'' << endl;

return isElseCorrect;

}

bool AstBuilder::isWhileCorrect(const string& line)

{

string patternStr = R"(^\s\*WHILE\s\*BE\s\*DO\s\*$)";

regex pattern(patternStr);

smatch match;

bool isWhileCorrect = regex\_match(line, match, pattern);

cout << "Is WHILE correct: " << isWhileCorrect <<" string: '" << line << '\'' << endl;

return isWhileCorrect;

}

GRAMMAR\_SYMBOL\_TYPE AstBuilder::determineGrammarSymbolType(string& line)

{

// block

if (line.find(PASCAL\_BEGIN) != string::npos) {

return GRAMMAR\_SYMBOL\_TYPE::*BLOCK*;

}

// Assignment

if (line.find(PASCAL\_ASSIGNMENT) != string::npos) {

bool isCorrect = isAssignmentCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::*ASSIGNMENT* : GRAMMAR\_SYMBOL\_TYPE::*ERROR*;

}

// if

if (line.find(PASCAL\_IF) != string::npos) {

bool isCorrect = isIfCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::*IF* : GRAMMAR\_SYMBOL\_TYPE::*ERROR*;

}

// else

if (line.find(PASCAL\_ELSE) != string::npos) {

bool isCorrect = isElseCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::*ELSE* : GRAMMAR\_SYMBOL\_TYPE::*ERROR*;

}

// while

if (line.find(PASCAL\_WHILE) != string::npos) {

bool isCorrect = isWhileCorrect(line);

return isCorrect ? GRAMMAR\_SYMBOL\_TYPE::*WHILE* : GRAMMAR\_SYMBOL\_TYPE::*ERROR*;

}

return GRAMMAR\_SYMBOL\_TYPE::*ERROR*;

}

bool AstBuilder::isOneLiner(const string& line)

{

return line.find(PASCAL\_ASSIGNMENT) != string::npos;

}

void AstBuilder::makeExpression(const vector<string>& parentExpression, size\_t index, vector<string>& returnVector)

{

if (parentExpression[index].find(PASCAL\_BEGIN) != string::npos) {

returnVector.push\_back(parentExpression[index]);

int foundBegins = 1;

index++;

while (index != parentExpression.size() && foundBegins != 0) {

if (parentExpression[index].find(PASCAL\_BEGIN) != string::npos) {

foundBegins++;

}

if (parentExpression[index].find(PASCAL\_END) != string::npos) {

foundBegins--;

}

returnVector.push\_back(parentExpression[index]);

index++;

}

return;

} else if (isOneLiner(parentExpression[index])) {

returnVector.push\_back(parentExpression[index]);

return;

} else {

returnVector.push\_back(parentExpression[index]);

return makeExpression(parentExpression, index + 1, returnVector);

}

}

Далее представлен исходный код файла main.cpp, в котором находится входная точка в программу и исходный код для печати дерева.

#include <iostream>

#include <iomanip>

#include <fstream>

#include <vector>

#include <list>

#include "AstBuilder.h"

#include "main.h"

using namespace std;

void print\_Tree(AstNode\* p, int level)

{

if (p)

{

if (level) {

cout << setw(level) << ' ';

}

switch (p->type)

{

case *NODE\_ASSIGNMENT*:

cout << ":=" << endl;

break;

case *NODE\_BLOCK*:

cout << "BLOCK" << endl;

break;

case *NODE\_IF*:

cout << "IF" << endl;

break;

case *NODE\_WHILE*:

cout << "WHILE" << endl;

break;

default:

cout << p->data << endl;

break;

}

for (AstNode\* node : p->list) {

print\_Tree(node, level + 4);

}

}

}

int main() {

string filePath = "test";

ifstream input(filePath);

if (!input.good()) {

cout << "Can't open source file" << endl;

exit(-1);

}

vector<string> lines;

string currentLine;

while (getline(input, currentLine)) {

if (!currentLine.empty()) {

lines.push\_back(currentLine);

}

}

AstBuilder builder;

AstNode\* root = builder.buildAstTree(lines);

cout << endl << endl;

print\_Tree(root, 0);

cout << endl << endl;

bool isCorrect = root != nullptr;

if (isCorrect) {

cout << "Source code is correct" << endl;

} else {

cout << "Source code is not correct" << endl;

}

return 0;

}

# 8-9 интеграция лексического анализатора и синтаксического анализатора

При трансляции исходный код претерпевает несколько последовательных преобразований. Так, сначала лексический анализатор разбивает текст программы на токены, список которых впоследствии используется для построения синтаксического дерева. Следует учесть, что элементами синтаксического дерева могут быть выражения, которые также должны быть представлены в виде древовидной структуры данных.

**Задание на лабораторную работу:** написать программу, объединяющую работу лексического и синтаксического анализаторов.

Для построения списка токенов был написан класс TokenListBuilder. Его исходный код представлен ниже. Для определения типа токена используются регулярные выражения. Типы токенов определены в перечислимом типе SYMBOL\_CLASS. Для хранения информации о токенах используется структура данных TOKEN. Она состоит из двух полей: тип токена и его значения. Значение представлено обхединением SYMBOL\_VALUE.

union SYMBOL\_VALUE {

float num;

bool boolean\_val;

char str\_val[64];

};

enum SYMBOL\_CLASS {

*ERROR*,

*NUMBER*, *BOOLEAN*,

*OPERATION\_MUL*, *OPERATION\_DIV*, *OPERATION\_ADD*, *OPERATION\_SUB*,

*OPERATION\_AND*, *OPERATION\_OR*, *OPERATION\_EQUALS*, *OPERATION\_NOT\_EQUALS*,

*OPEN\_BRACKET*, *CLOSE\_BRACKET*, *SEMICOLON*,

*COMMAND\_WHILE*, *COMMAND\_IF*, *COMMAND\_ELSE*, *COMMAND\_DO*, *COMMAND\_THEN*,

*COMMAND\_BEGIN*, *COMMAND\_END*, *COMMAND\_ASSIGNMENT*,

*ID*

};

#include "TokenListBuilder.h"

SYMBOL\_CLASS TokenListBuilder::determineSymbolClass(string &word) {

map<SYMBOL\_CLASS, string> classToRegex;

classToRegex[*NUMBER*] = R"(((-?\d+(\.\d+)?e[-+]\d+)|(-?\d+\.\d+)|(-?\d+)))";

classToRegex[*BOOLEAN*] = R"(TRUE|FALSE)";

classToRegex[*ID*] = R"([a-zA-Z](\d|[a-zA-Z])\*)";

classToRegex[*OPERATION\_ADD*] = R"(\+)";

classToRegex[*OPERATION\_MUL*] = R"(\\*)";

classToRegex[*OPERATION\_DIV*] = R"(/)";

classToRegex[*OPERATION\_SUB*] = R"(-)";

classToRegex[*OPERATION\_AND*] = R"(AND)";

classToRegex[*OPERATION\_OR*] = R"(OR)";

classToRegex[*OPERATION\_EQUALS*] = R"(==)";

classToRegex[*OPERATION\_NOT\_EQUALS*] = R"(!=)";

classToRegex[*OPEN\_BRACKET*] = R"(\()";

classToRegex[*CLOSE\_BRACKET*] = R"(\))";

classToRegex[*SEMICOLON*] = R"(;)";

classToRegex[*COMMAND\_ASSIGNMENT*] = R"(:=)";

classToRegex[*COMMAND\_WHILE*] = R"(WHILE)";

classToRegex[*COMMAND\_DO*] = R"(DO)";

classToRegex[*COMMAND\_BEGIN*] = R"(BEGIN)";

classToRegex[*COMMAND\_END*] = R"(END)";

classToRegex[*COMMAND\_IF*] = R"(IF)";

classToRegex[*COMMAND\_THEN*] = R"(THEN)";

classToRegex[*COMMAND\_ELSE*] = R"(ELSE)";

for (const auto& cToReg : classToRegex) {

regex reg(cToReg.second);

smatch match;

bool isRegRight = regex\_match(word, match, reg);

if (isRegRight) {

return cToReg.first;

}

}

return SYMBOL\_CLASS::*ERROR*;

}

vector<TOKEN> TokenListBuilder::parseInput(vector<string> words) {

vector<TOKEN> returnVector;

auto spaceRegex = regex(R"(\s+)");

smatch match;

for (auto & word : words) {

if (regex\_match(word, match, spaceRegex) || word.empty()) {

continue;

}

TOKEN token;

SYMBOL\_CLASS symbolClass = determineSymbolClass(word);

if (symbolClass == SYMBOL\_CLASS::*ERROR*) {

cout << "Can't determine type of token: " << word;

exit(-1);

}

token.type = symbolClass;

switch (symbolClass) {

case *NUMBER*: {

token.sym.num = atof(word.c\_str());

break;

}

case *BOOLEAN*: {

token.sym.boolean\_val = word == "TRUE";

break;

}

default: {

memcpy(token.sym.str\_val, word.c\_str(), sizeof(char) \* 64);

break;

}

}

returnVector.push\_back(token);

}

return returnVector;

}

vector<TOKEN> TokenListBuilder::buildTokenList(const string& fileName) {

fstream input(fileName);

if (!input.good()) {

cout << "Can't open file!";

exit(-1);

}

vector<string> words;

while (!input.eof()) {

string tmp;

input >> tmp;

words.push\_back(tmp);

}

vector<TOKEN> tokenList = parseInput(words);

return tokenList;

}

Для построения абстрактного синтаксического дерева был реализован класс AstBuilder. В нем также определена структура данных AstNode — узел АСТ дерева. Она состоит из трех полей: тип узла, список дочерних элементов, и токен. Последнее поле используется для реализации деревьев выражений. Тип узла определяется перечислением AST\_NODE\_TYPE. Далее приведена реализация класса AstBuilder. Основная часть работы происходит в рекурсивной процедуре checkAndBuildTree, принимающей список токенов, из которых надо построить дерево и узел, в который необходимо записать результат.

enum AST\_NODE\_TYPE {

*BLOCK*, *WHILE*, *IF*, *ASSIGNMENT*, *ARITHMETIC\_EXPR*, *BOOL\_EXPR*, *ID\_AST*

};

struct AstNode {

AST\_NODE\_TYPE type;

vector<AstNode\*> children;

TOKEN exprToken;

};

#include "AstBuilder.h"

using namespace std;

bool AstBuilder::checkAndBuildExprTree(vector<TOKEN> tokens, AstNode \*&root) {

vector<TOKEN> rpn;

stack<AstNode\*> stOperations;

try {

rpn = buildExprRpn(tokens);

} catch (exception &e) {

return false;

}

root = new AstNode();

root->exprToken = rpn.back();

AstNode\* prevNode = root;

prevNode->children.push\_back(nullptr);

prevNode->children.push\_back(nullptr);

for (long long i = rpn.size() - 2; i >= 0; i--) {

auto currentNode = new AstNode();

currentNode->exprToken = rpn[i];

while (prevNode->children[0] != nullptr && prevNode->children[1] != nullptr) {

prevNode = stOperations.top();

stOperations.pop();

}

if (prevNode->children[1] == nullptr) {

prevNode->children[1] = currentNode;

} else {

prevNode->children[0] = currentNode;

}

if (rpn[i].type != SYMBOL\_CLASS::*NUMBER*) {

stOperations.push(prevNode);

prevNode = currentNode;

prevNode->children.push\_back(nullptr);

prevNode->children.push\_back(nullptr);

}

}

root->exprToken = rpn.back();

return true;

}

bool AstBuilder::checkAndBuildBooleanExprTree(vector<TOKEN> tokens, AstNode \*&root) {

vector<TOKEN> rpn;

stack<AstNode\*> stOperations;

try {

rpn = buildBoolExprRpn(tokens);

} catch (exception &e) {

return false;

}

root = new AstNode();

root->exprToken = rpn.back();

AstNode\* prevNode = root;

prevNode->children.push\_back(nullptr);

prevNode->children.push\_back(nullptr);

for (long long i = rpn.size() - 2; i >= 0; i--) {

auto currentNode = new AstNode();

currentNode->exprToken = rpn[i];

while (prevNode->children[0] != nullptr && prevNode->children[1] != nullptr) {

prevNode = stOperations.top();

stOperations.pop();

}

if (prevNode->children[1] == nullptr) {

prevNode->children[1] = currentNode;

} else {

prevNode->children[0] = currentNode;

}

if (rpn[i].type != SYMBOL\_CLASS::*NUMBER*) {

stOperations.push(prevNode);

prevNode = currentNode;

prevNode->children.push\_back(nullptr);

prevNode->children.push\_back(nullptr);

}

}

root->exprToken = rpn.back();

return true;

}

vector<TOKEN> AstBuilder::buildBoolExprRpn(vector<TOKEN> &tokens) {

vector<SYMBOL\_CLASS> allowedTokens;

allowedTokens.push\_back(SYMBOL\_CLASS::*OPEN\_BRACKET*);

allowedTokens.push\_back(SYMBOL\_CLASS::*CLOSE\_BRACKET*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_AND*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_OR*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_EQUALS*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_NOT\_EQUALS*);

allowedTokens.push\_back(SYMBOL\_CLASS::*BOOLEAN*);

for (auto &token : tokens) {

auto isTokenAllowed = false;

for (auto &allowedToken : allowedTokens) {

if (token.type == allowedToken) {

isTokenAllowed = true;

break;

}

}

if (!isTokenAllowed) {

throw runtime\_error("Incorrect expression");

}

}

map<SYMBOL\_CLASS, int> classToPriority;

classToPriority[*OPERATION\_AND*] = 1;

classToPriority[*OPERATION\_OR*] = 1;

classToPriority[*OPERATION\_EQUALS*] = 1;

classToPriority[*OPERATION\_NOT\_EQUALS*] = 1;

classToPriority[*OPEN\_BRACKET*] = 0;

vector<TOKEN> result;

stack<TOKEN> st;

for (TOKEN currentToken : tokens) {

switch (currentToken.type) {

case SYMBOL\_CLASS::*BOOLEAN*: {

result.push\_back(currentToken);

break;

}

case *OPEN\_BRACKET*: {

st.push(currentToken);

break;

}

case *CLOSE\_BRACKET*: {

while (st.top().type != *OPEN\_BRACKET*) {

result.push\_back(st.top());

st.pop();

if (st.empty()) {

throw out\_of\_range("Invalid expression");

}

}

st.pop();

break;

}

default: {

if (currentToken.type != *OPERATION\_AND*

&& currentToken.type != *OPERATION\_OR*

&& currentToken.type != *OPERATION\_EQUALS*

&& currentToken.type != *OPERATION\_NOT\_EQUALS*) {

cout << "Incorrect expression";

exit(-1);

}

while (!st.empty() && (classToPriority[currentToken.type] <= classToPriority[st.top().type])) {

result.push\_back(st.top());

st.pop();

}

st.push(currentToken);

}

}

}

while (!st.empty()) {

result.push\_back(st.top());

st.pop();

}

return result;

}

vector<TOKEN> AstBuilder::buildExprRpn(vector<TOKEN> &tokens) {

vector<SYMBOL\_CLASS> allowedTokens;

allowedTokens.push\_back(SYMBOL\_CLASS::*OPEN\_BRACKET*);

allowedTokens.push\_back(SYMBOL\_CLASS::*CLOSE\_BRACKET*);

allowedTokens.push\_back(SYMBOL\_CLASS::*NUMBER*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_DIV*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_MUL*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_ADD*);

allowedTokens.push\_back(SYMBOL\_CLASS::*OPERATION\_SUB*);

for (auto &token : tokens) {

auto isTokenAllowed = false;

for (auto &allowedToken : allowedTokens) {

if (token.type == allowedToken) {

isTokenAllowed = true;

break;

}

}

if (!isTokenAllowed) {

throw runtime\_error("Incorrect expression");

}

}

map<SYMBOL\_CLASS, int> classToPriority;

classToPriority[*OPERATION\_ADD*] = 1;

classToPriority[*OPERATION\_SUB*] = 1;

classToPriority[*OPERATION\_MUL*] = 2;

classToPriority[*OPERATION\_DIV*] = 2;

classToPriority[*OPEN\_BRACKET*] = 0;

vector<TOKEN> result;

stack<TOKEN> st;

for (TOKEN currentToken : tokens) {

switch (currentToken.type) {

case SYMBOL\_CLASS::*NUMBER*: {

result.push\_back(currentToken);

break;

}

case *OPEN\_BRACKET*: {

st.push(currentToken);

break;

}

case *CLOSE\_BRACKET*: {

while (st.top().type != *OPEN\_BRACKET*) {

result.push\_back(st.top());

st.pop();

if (st.empty()) {

throw out\_of\_range("Invalid expression");

}

}

st.pop();

break;

}

default: {

if (currentToken.type != *OPERATION\_SUB*

&& currentToken.type != *OPERATION\_DIV*

&& currentToken.type != *OPERATION\_ADD*

&& currentToken.type != *OPERATION\_MUL*) {

cout << "Incorrect expression";

exit(-1);

}

while (!st.empty() && (classToPriority[currentToken.type] <= classToPriority[st.top().type])) {

result.push\_back(st.top());

st.pop();

}

st.push(currentToken);

}

}

}

while (!st.empty()) {

result.push\_back(st.top());

st.pop();

}

return result;

}

void AstBuilder::deleteTree(AstNode\* root)

{

if (root == nullptr) {

return;

}

for (AstNode\* node : root->children) {

deleteTree(node);

}

delete root;

}

void AstBuilder::makeOneLiner(

const vector<TOKEN>& parentExpression,

size\_t index,

vector<TOKEN>& returnVector

) {

while (parentExpression[index].type != SYMBOL\_CLASS::*SEMICOLON*) {

returnVector.push\_back(parentExpression[index++]);

}

returnVector.push\_back(parentExpression[index]);

}

bool isOneLiner(const TOKEN& token) {

// Only assignment can be oneliner. It's always starts with ID

return token.type == SYMBOL\_CLASS::*ID*;

}

// Is used for making content of BEGIN-END block of every

// complex expression (IF, ELSE, WHILE)

void AstBuilder::makeBlockContent(

const vector<TOKEN>& parentExpression,

size\_t index,

vector<TOKEN>& returnVector

) {

if (parentExpression[index].type == SYMBOL\_CLASS::*COMMAND\_BEGIN*) {

returnVector.push\_back(parentExpression[index]);

int foundBegins = 1;

index++;

while (index != parentExpression.size() && foundBegins != 0) {

if (parentExpression[index].type == SYMBOL\_CLASS::*COMMAND\_BEGIN*) {

foundBegins++;

}

if (parentExpression[index].type == SYMBOL\_CLASS::*COMMAND\_END*) {

foundBegins--;

}

returnVector.push\_back(parentExpression[index]);

index++;

}

} else if (isOneLiner(parentExpression[index])) {

makeOneLiner(parentExpression, index, returnVector);

return;

} else {

returnVector.push\_back(parentExpression[index]);

return makeBlockContent(parentExpression, index + 1, returnVector);

}

}

bool AstBuilder::checkAndBuildTree(vector<TOKEN> tokens, AstNode \*&root) {

bool isCorrect = true;

std::size\_t i = 0;

while (i < tokens.size()) {

auto\* result = new AstNode();

if (root != nullptr) {

root->children.push\_back(result);

} else {

root = result;

}

TOKEN currentToken = tokens[i];

switch (currentToken.type) {

case SYMBOL\_CLASS::*ERROR*: {

cout << "Bad token: " << currentToken.sym.str\_val;

break;

}

case SYMBOL\_CLASS::*COMMAND\_BEGIN*: {

vector<TOKEN> blockContent;

makeBlockContent(tokens, i, blockContent);

i += blockContent.size() + 1;

// Remove first BEGIN and last END (it's belong to parent block)

blockContent.erase(blockContent.begin());

blockContent.erase(blockContent.begin() + blockContent.size() - 1);

result->type = AST\_NODE\_TYPE::*BLOCK*;

checkAndBuildTree(blockContent, root);

break;

}

case SYMBOL\_CLASS::*COMMAND\_WHILE*: {

result->type = AST\_NODE\_TYPE::*WHILE*;

vector<TOKEN> boolExpr;

if (i == tokens.size() - 1) {

return false;

}

while (tokens[++i].type != SYMBOL\_CLASS::*COMMAND\_DO*) {

boolExpr.push\_back(tokens[i]);

if (i == tokens.size() - 1) {

return false;

}

}

auto exprRoot = new AstNode();

checkAndBuildBooleanExprTree(boolExpr, exprRoot);

if (exprRoot == nullptr) {

return false;

}

exprRoot->type = AST\_NODE\_TYPE::*BOOL\_EXPR*;

result->children.push\_back(exprRoot);

result->children.push\_back(nullptr);

vector<TOKEN> whileContent;

makeBlockContent(tokens, i + 1, whileContent);

isCorrect = checkAndBuildTree(whileContent, result->children.back());

if (!isCorrect) { return false; }

i += whileContent.size() + 1;

break;

}

// Assignment starts with ID

case SYMBOL\_CLASS::*ID*: {

result->type = AST\_NODE\_TYPE::*ASSIGNMENT*;

auto variable = new AstNode();

variable->type = AST\_NODE\_TYPE::*ID\_AST*;

result->children.push\_back(variable);

result->children.push\_back(nullptr);

// Get token after assignment ':='

i += 2;

vector<TOKEN> exprTokens;

while (tokens[i].type != SYMBOL\_CLASS::*SEMICOLON*) {

exprTokens.push\_back(tokens[i++]);

}

isCorrect = checkAndBuildExprTree(exprTokens, result->children.back());

if (!isCorrect) { return false; }

result->children.back()->type = AST\_NODE\_TYPE::*ARITHMETIC\_EXPR*;

break;

}

case SYMBOL\_CLASS::*COMMAND\_IF*: {

result->type = AST\_NODE\_TYPE::*IF*;

vector<TOKEN> boolExprTokens;

if (i == tokens.size() - 2) {

return false;

}

// Get boolean expression

while (tokens[++i].type != SYMBOL\_CLASS::*COMMAND\_THEN*) {

if (i == tokens.size() - 2) {

return false;

}

boolExprTokens.push\_back(tokens[i]);

}

result->children.push\_back(nullptr);

isCorrect = checkAndBuildBooleanExprTree(boolExprTokens, result->children.back());

if (!isCorrect) { return false; }

result->children.back()->type = AST\_NODE\_TYPE::*BOOL\_EXPR*;

// Get THEN content

result->children.push\_back(nullptr);

vector<TOKEN> thenContent;

makeBlockContent(tokens, i + 1, thenContent);

isCorrect = checkAndBuildTree(thenContent, result->children.back());

if (!isCorrect) { return false; }

i += thenContent.size() + 1;

result->children.push\_back(nullptr);

if (tokens[i].type == SYMBOL\_CLASS::*COMMAND\_ELSE*) {

vector<TOKEN> elseContent;

makeBlockContent(tokens, i + 1, elseContent);

isCorrect = checkAndBuildTree(elseContent, result->children.back());

if (!isCorrect) { return false; }

i += elseContent.size() + 1;

}

break;

}

default: {

}

}

i++;

}

return isCorrect;

}

AstNode\* AstBuilder::buildAstTree(vector<TOKEN>& tokens) {

AstNode\* result = nullptr;

if (!checkAndBuildTree(tokens, result)) {

deleteTree(result);

result = nullptr;

}

return result;

}